Automatic Functional Verification of Memory Oriented Global Source Code Transformations

K.C. Shashidhar^{1,2}, M. Bruynooghe², F. Catthoor^{1,2} and G. Janssens²

¹IMEC vzw, Kapeldreef 75, B-3001 Heverlee, Belgium

{kodambal,catthoor}@imec.be

²Faculteit Toegepaste Wetenschappen, Katholieke Universiteit Leuven, Belgium

{maurice,gerda}@cs.kuleuven.ac.be

Abstract

In this paper, we present a fully automatic technique to verify an important class of optimizing program transformations applied to reduce accesses to the data memory. These are prevalent while developing software for power and performance-efficient embedded multimedia systems. The verification of the transformations relies on an automatic proof of functional equivalence of the initial and the transformed program functions. It is based on extracting and reasoning on the polyhedral models representing the dependencies between the elements of the output and the input variables, which are preserved under the transformations considered. If the verification reports failure, the technique also identifies the errors and their location in the function, hence providing an effective means to debug the transformed program function.

1 Introduction

Design of power and performance-efficient embedded multimedia systems requires a prolonged optimization phase at all abstraction levels, particularly at the executable specification level. In multimedia systems, the workhorses of the application are the routines implementing the signal processing algorithms. Optimization of these processing routines is the key to an efficient implementation of the system. They have to be highly tuned to the custom designed memory and processor architecture in order to meet the design constraints.

Experience shows that presently available compilers are not effective for optimizing the program accesses to the data memory. Based on the knowledge of the platform and memory access behavior of the algorithm, at present, experienced designers are often applying global source code transformations to arrive at a better final implementation. For various reasons (mainly, flexibility), such transformations are being applied either manually or with ad-hoc tools. But this comes at a price, namely, the additional overhead on the verification phase to check the *correctness of the application of such transformations*. A transformation verification tool that is able to verify the equivalence

between the transformed version and the original is required to reduce this overhead. Our work addresses the core of this requirement.

In this paper, we focus on the automatic functional verification of an important class of transformations applied in the context of custom memory management for embedded systems, viz., loop transformations, that improve the locality and regularity of the memory accesses and expression propagation transformations, that remove the redundant accesses and also enable a good schedule for the accesses at a high level [3].

The outline of the paper: Sections 2 and 3 delineate the addressed problem with an example and discussion of related work, Sections 4, 5 and 6 present our verification technique, support for providing error diagnosis and experimental results and Section 7 provides a justification for the assumptions made and conclusions.

2 Problem Statement

The problem that is addressed here is to automatically verify the input-output functional equivalence of two sequential imperative program functions, where one has been obtained from the other by applying: (1) Loop transformations; and/or (2) Expression propagation transformations on the program, *intraprocedurally*. An assumption is that the program functions have the property that each memory element that is assigned a value is done so only once (i.e., they are in *single-assignment form*).

The loop transformations modify only the schedule of the assignment statements in the loop body without violating the data-flow order (i.e., only values that have already been produced are consumed). We handle all loop transformations and do not distinguish between structure-preserving (viz., interchange, skewing, reversal, bumping) and structure-modifying (viz., distribution, fusion, tiling) loop transformations.

The expression propagation transformations introduce or eliminate intermediate variables by propagating expressions or sub-expressions in the assignment statements in the function. To ensure that this does not result in an *algebraic* transformation, we restrict the propagated expressions to either function calls

<pre> for (i=0;i<256;i++) for (j=1;j<128;j++) a[i][j] = f(b[i+2][j-1]) + g(c[i+1]); </pre> <p style="text-align: center;">(a)</p>	<pre> for (i=0;i<256;i++){ tmp[i] = g(c[i+1]); for (j=1;j<128;j++) a[i][j] = f(b[i+2][j-1]) + tmp[i]; } </pre> <p style="text-align: center;">(b)</p>	<pre> for (i=0;i<256;i++){ tmp1[i] = c[i+1]; tmp2[i] = g(tmp1[i]); for (j=1; j<128; j++){ buf[i][j] = f(b[i+2][j-1]); a[i][j] = buf[i][j] + tmp2[i]; } } </pre> <p style="text-align: center;">(c)</p>
--	--	---

Figure 1: Simple example to illustrate expression propagation transformations.

or parenthesized expressions. For example, Figure 1 shows small, functionally equivalent code fragments, which are obtained from one another by expression propagation.

The transformations applied on the function that is being verified are restricted to only those that fall into one of the categories mentioned above. The verification is required to be done *oblivious* of the particular transformations of the mentioned class that were applied and the order of their application.

3 Related Work

The hardware and software verification literature is very vast, but, it seems that the available methods cannot be used or adapted to solve the equivalence checking problem for program functions with many array variables and `for`-loops. Some solutions are available based mainly on program transformation systems coupled with provers. But we believe they have not been successfully deployed in design environments in the industry owing to either their inflexibility or lack of skill in their use. Simulation-based testing is still the most commonly employed method of checking the correctness of transformed specifications to the original specification (the so-called golden reference) in the industry. But, testing is very time consuming and yet, often incomplete.

With the same notion of input-output equivalence as ours, in [5], an automatic approach is presented to compare structurally similar assembly-language routines for DSPs. It is based on symbolic simulation and use of an automated decision procedure. Importantly, they require that the loops are unrolled. In contrast, we check equivalence of source-language routines, where, code fragments under loops are our main concern and we do not require unrolling of the loops. Unrolling the loops (typically with large bounds) is practically infeasible, in our target domain, as it explodes the code size and the work needed to normalize the routines under the expression propagation transformations is prohibitively time consuming. On the other hand, we are not able to check algebraic equalities and, at present, we forbid such transformations. We would like to remark here that we hold the same arguments as provided in [5] towards related work found in formal verification literature and the practical relevance of a *safe*, but, less general method, as presented in this paper, to address the need for auto-

matic verification in a particular design context.

The work on translation validation [7, 11], provides another approach in a related but different context. They verify optimizations applied by an optimizing compiler during translation, relying on additional information generated by the compiler and heuristics/special analysis to infer a simulation relation between the source and the target codes. We target transformations applied in the system level design context, where we often do not have any information about the applied transformations.

In recent work [10], we have developed a method for verifying loop and data reuse transformations (a trivial form of expression propagation, where only intermediate copies are introduced). It was based on past work at IMEC [9, 4]. In the work presented here, we have significantly generalized that method to include a general case of expression propagation. The generalization is such that it offers a unified one-pass method for all transformations as opposed to a case by case checking.

In parallel to our work, a system similar to ours has been recently developed [2], mainly motivated by the automatic algorithm recognition problem. This system also takes two polyhedral models (systems of affine recurrence equations) and checks whether they are equal with respect to the inputs and outputs. We believe the distinction of our work is in the underlying mechanics of our equivalence checker and our stress on the support for error diagnostics. Our tool makes use of an in-house tool which takes source code (presently in the C programming language) and extracts the polyhedral model with annotations linking it to the assignment statements in the function. The equivalence checker maintains these annotations during the checking process and hence, upon failure, is able to provide useful diagnostics to locate the errors and effectively debug the transformed program. For a further discussion on related work we refer the reader to [10].

4 Verification Technique

The technique is an implementation of the scheme shown in Figure 2. Given the initial and transformed function pair, the polyhedral models are extracted using static analysis. These models capture the relations between the elements of the array variables. The models are input to the equivalence checker which identifies

Figure 2: Transformation verification scheme.

the required correspondences between the two models and determines their equivalence. In this, it relies on the OMEGA calculator [6] for operations on integer sets and tuple relations. The checker is conservative. Therefore, if it succeeds, the two program functions are functionally equivalent. If it fails, it provides the diagnostics to locate and debug the error.

4.1 Polyhedral Model of a Function

The polyhedral models are formulae that encode affine constraints on integer variables, logical connectives and quantifiers, also called *Presburger formulae*. These formulae can symbolically represent the dependency relation between the elements of the defined variable and the operand variables in each of the assignment statements in a function.

For example, the code fragment (a) in Figure 1 has a single assignment statement, with one defined variable (`a[[]]`) and two operand variables (`b[[]]` and `c[[]]`). The polyhedral model for this fragment captures the two relationships (dependency mappings), i.e., from the defined variable to each of the operand variables, in two integer tuple relations as below:

$$\begin{aligned}
 \mathbf{M}_{ab} &:= \{ [d_1, d_2] \rightarrow [d_3, d_4] \mid d_1 = i \wedge d_2 = j \\
 &\quad \wedge d_3 = i + 2 \wedge d_4 = j - 1 \wedge [i, j] \in \mathbf{D} \} \\
 \mathbf{M}_{ac} &:= \{ [d_1, d_2] \rightarrow [d_3] \mid d_1 = i \wedge d_2 = j \\
 &\quad \wedge d_3 = i + 1 \wedge [i, j] \in \mathbf{D} \}
 \end{aligned}$$

where,

$$\mathbf{D} := \{ [i, j] \mid 0 \leq i < 256 \wedge 1 \leq j < 128 \wedge [i, j] \in \mathbb{Z}^2 \}$$

The above model additionally contains an annotation, noting the signature of the computation in the assignment statement, i.e., $(f(V)) + (g(V))$.

4.2 Equivalence Checking Procedure

Before we begin the equivalence checking, as a preparatory step, we first check whether the array access ordering is correct in each of the program functions separately. By this we mean that each variable element that is read in an assignment statement has either been already defined or is an input variable. This is relatively easy to check as our program functions are in single-assignment form. In order to ensure this condition, we check that for any given outermost loop statement, all the elements that are read by the assignment statements inside it (at any nesting depth) are either:

- (i) a subset of the union of all the elements of the input variables and all the elements of the defined variables in the preceding statements in the program function; or
- (ii) have been written by an assignment statement in an earlier iteration of the same outermost loop statement (i.e., we have only the so-called *positive data dependences*).

This is a regular check in most array data-flow analysis methods (cf. [1]). Our checker calls an implementation of the above condition in an in-house array data-flow analysis tool. The two program functions are checked for equivalence only when this condition has been met.

Intuitively, for the two functions to be functionally equivalent with respect to the input and output variables, under the considered transformations, it is *sufficient* that the following two conditions hold:

- I. The computation in the corresponding data-flow dependency chains is equivalent; and
- II. The dependency relations between the elements of the output variables and the input variables are identical in both functions.

The equivalence checker essentially verifies the above two conditions. The checking process always maintains correspondences between the two functions that should hold for the equivalence proof to succeed. These correspondences are captured in proof obligations. We define a proof obligation as a two element tuple: an element for each of the two functions. Each element comprises of a variable identifier, a dependency relation from the output variable to that variable and the signature of the computation which defines the variable. Additionally, it also maintains some book-keeping information.

A sketch of how the equivalence checker proceeds with the proof is provided in Algorithm 1. An initial set of proof obligations is composed of the necessary correspondences that should hold for the equivalence to hold. This is trivially the one-to-one correspondence between the output variables of the two functions. They are identified by their names, which have

Algorithm 1 Outline of the equivalence checker

```

function PromoteObligation( $p$ )
  Input: a proof obligation  $p$ , that is to be promoted
  Output: if successful, return True with new obligations
           else return False with diagnostics

  if RecurrentObligation( $p$ ) then
    Handle recurrence
  end if
  if CompatibleSignatures() then
    Peel signatures
    Update relations
    Construct newObligations
  else
    return (False, diagnostics)
  end if
  return (True, newObligations)
end function

```

```

CheckerEngine()
Input: polyhedral models of the two functions
Output: if they are equivalent, return True
           else return False with diagnostics

```

```

Initialize the proof obligations list
while proof obligations list not empty do
  Pick a proof obligation  $p$  from the list
  if TerminalObligation( $p$ ) then
    if not EquivalentObligation( $p$ ) then
      return (False, diagnostics)
    end if
  else
    if PromoteObligation( $p$ ) then
      Update proof obligations list
      Update diagnostics
    else
      return (False, diagnostics)
    end if
  end if
end while
return True

```

to match. The initial obligations are then promoted up the corresponding data-flow dependency chains of the two functions until the input variables are reached.

The promotion is done in a stepwise manner, at each step accounting for matching signatures of computation in both the functions and updating the dependency relation. All operators and functions are uninterpreted and, at present, we do not support algebraic data-flow transformations (those involving the commutativity, associativity, ... properties of the applied functions in the assignment statements). Hence, it suffices to check the equivalence of computation by pattern matching the signatures of the computation at each step. If the signatures do not exactly match, they should at least be *compatible*. Compatible signatures are those which contain either components with matching outermost signatures or components with as yet unmatched signatures, but no non-matching signatures. For example, $(f(g(V)))$ is compatible with $(f(V))$ and (V) , but not with $(f(h(V)))$ or $(h(g(V)))$. Compatible signatures with matching outermost signatures can be *peeled* to fork new proof obligations for the checking to continue.

Data-flow may have *recurrences*, that is, a set of statements may read values written by themselves in earlier iterations. Clearly, it is inefficient to step through each instance of the recurrence. The checker avoids this by stepping over the recurrence by com-

Figure 3: The data-flow graphs for the example.

puting the elements at the end of the recurrence. This requires a so-called *transitive closure* operation on the tuple relation and is exactly computable only under certain conditions. In practice, however, these conditions usually hold.

For a dependency chain, the promotion of obligations stops when both the variables in the obligation are input variables. The dependency relations are checked for equivalence at this point. This is repeated until all proof obligations are exhausted. For lack of space, we omit further discussion of the details of Algorithm 1.

4.3 Illustration on a Simple Example

Consider the simple code fragments (b) and (c) in Figure 1. The data-flow graphs of the two functions are as shown in Figure 3. The equivalence checking process is as shown in Figure 4. The text in the box is the verbose output from our tool.

The individual steps of the process can also be seen in the progression of proof obligations (dotted edges) between the data-flow graphs in Figure 3. The checking starts with the only known initial correspondence, i.e., between the array variable a in each function, and proceeds by identifying the next correspondences. Notice that at each step, consecutive tuple relations in the data-flow are joined in order to update the dependency relation from the output variable up to the current point in the data-flow chain. It terminates when the correspondences reach b and c , the input variables.

5 Error Diagnostics

The checker may fail to prove equivalence under two situations, corresponding to our two sufficient conditions. In either case, the checker provides diagnostics for failure.

```

initial proof obligations:
---(1)---
['a', ' {[a0,a1] -> [a0,a1]: 0 <= a0 <= 255 && 1 <= a1 <= 127}', '(f(V)) + (V)']
['a', ' {[a0,a1] -> [a0,a1]: 0 <= a0 <= 255 && 1 <= a1 <= 127}', '(V) + (V)']

new proof obligations:
---(1-1)---
['tmp', ' {[a0,a1] -> [a0]: 0 <= a0 <= 255 && 1 <= a1 <= 127}', '(g(V))']
['tmp2', ' {[a0,a1] -> [a0]: 0 <= a0 <= 255 && 1 <= a1 <= 127}', '(g((V)))']

---(1-2)---
['a->(1)', ' {[a0,a1] -> [a0,a1]: 0 <= a0 <= 255 && 1 <= a1 <= 127}', '(f(V))']
['buf', ' {[a0,a1] -> [a0,a1]: 0 <= a0 <= 255 && 1 <= a1 <= 127}', '(f(V))']

new proof obligations:
---(1-1-1)---
['c', ' {[a0,a1] -> [a0+1]: 0 <= a0 <= 255 && 1 <= a1 <= 127}', '']
['tmp1', ' {[a0,a1] -> [a0]: 0 <= a0 <= 255 && 1 <= a1 <= 127}', '(V)']

new proof obligations:
---(1-1-1-1)---
['c', ' {[a0,a1] -> [a0+1]: 0 <= a0 <= 255 && 1 <= a1 <= 127}', '']
['c', ' {[a0,a1] -> [a0+1]: 0 <= a0 <= 255 && 1 <= a1 <= 127}', '']

----> TerminalObligation
-----> EquivalentObligation

new proof obligations:
---(1-2-1)---
['b', ' {[a0,a1] -> [a0+2,a1-1]: 0 <= a0 <= 255 && 1 <= a1 <= 127}', '']
['b', ' {[a0,a1] -> [a0+2,a1-1]: 0 <= a0 <= 255 && 1 <= a1 <= 127}', '']

----> TerminalObligation
-----> EquivalentObligation

Equivalence checking succeeded!

```

Figure 4: Illustration of the equivalence checking process. In the obligation (1-2), 'a->(1)' refers to the 1st operand in the statement defining a[] [].

The first situation arises when the computations (signatures) on the two corresponding sides do not match. In this case, the checker stops and points to the exact location of the mismatching statements in the two functions.

The second situation arises when the number of elements, that should correspond on either side, do not match. In this case, the checker stops and reports the compatible assignment statements on both sides that should collectively account for all the elements and the difference in the range of elements that causes the mismatch. The number of assignment statements depends on the program functions in question. Typically, we expect this diagnostic to be sufficient for debugging. A pathological case arises when such a mismatch shows up only after reaching the inputs. In such a case, the checker reports the output variables in question and the difference in the range of elements on either side and their range. The usefulness of such a diagnostic, however, depends on the length of the data-flow graph.

6 Experimental Results

We have implemented the equivalence checking procedure in a prototype tool developed in the Python programming language. The verification times that we observe on some real life examples are in the order of few seconds. As mentioned earlier, the checker does not unroll the loops, therefore, the verification time is independent of the size of the loop bounds.

DFG size (Depth, Width)	Lines of code (Ver.1, Ver.2)	Time taken (in seconds)
(4, 16)	(140, 50)	20
(4, 24)	(220, 110)	28
(4, 32)	(320, 230)	40
(8, 16)	(250, 170)	44
(8, 24)	(380, 210)	62
(8, 32)	(520, 240)	86

Table 1: Times required for verification (on a Pentium 4, 1.7 GHz).

A characterization of the problem size is the length the constraint expressions in dependency relations in the functions. The verification time is mainly dependent on this. This is due to the fact that the operations on the polyhedral models involve determining validity in the Presburger arithmetic and it has a super-exponential complexity in the length of the constraint expression. However, this does not pose a problem in practice. In typical real application kernels, the length of the constraint expressions in dependency relations are reasonable. Moreover, our equivalence checker keeps the expressions small by slicing the obligations into small corresponding pairs and also by simplifying the expressions at every step. Also, we use OMEGA calculator, which is an efficient solver in practice [8]. Finally, if the constraint length indeed poses a problem, the user can help the checker by providing information about known correspondences between intermediate variables in the two functions.

Another characterization of the problem size is the nature of the data-flow in the program function. The depth and the width (number of leaves) of the data-flow graph (DFG), determines the load on the checker. A program function will have a forest of DFGs and the total verification time, in the worst case, is the sum of the times taken for the corresponding DFGs. But, in practice, the DFGs usually overlap and the checker exploits this by establishing equivalence only once for a given variable pair.

We believe that our method is scalable to large function sizes in practice. This is substantiated by the examples in Table 1, where we have used successively larger DFGs constructed with realistic complexity in control flow and variable index expressions.

Our verification tool has detected and localized many errors introduced in the codes in the table above. Also, a preliminary version of our tool has detected an error in the transformed USVD code that was ultimately traced to the code generator of a prototype loop transformation tool.

7 Discussion

The equivalence checking problem is, in general, well-known to be undecidable. This means that the class of programs that can be checked is fundamen-

tally restricted. Hence, assumptions with respect to the permissible programs and transformations are unavoidable. For the functions that are input to the tool, we have assumed the following:

- Each variable (other than the iterators) is written only once, i.e., the program is in single-assignment form. Any scalar variables that are present are deemed to be array variables with a single element.
- The indices for all references to arrays and any conditionals present can be statically determined at compile time, i.e., they are not input-dependent. Consequently, **while**-loops are not allowed.
- The index expressions for references to arrays, any conditionals in loop body and the loop execution bounds are all piece-wise affine functions (including **mod**, **div**, **max**, **min**, **floor** and **ceil** operators) of the controlling loop iterators.
- No pointer references are present in the function. Consequently, dynamic memory allocation is not allowed.
- The output and input variables (the interface variables of the function) are not renamed in the transformed function. (This correspondence forms the basis for the start and termination of equivalence checking.)

The above assumptions are, however, reasonable in the design context where the addressed verification problem arises. The transformations are often applied based on information from some bottleneck analysis and estimation tools. These tools can do a better job when the source code being analyzed is made less complex in a preprocessing stage. This is typical in a high level design exploration methodology, like, for example, in the Data Transfer and Storage Exploration (DTSE) methodology [3] developed at IMEC. Ongoing work is focussed on developing tools to perform such source code preprocessing like code conversion to dynamic single-assignment form, pointer to array conversion, etc. We expect this will allow us to handle more general program functions.

Transformations that our verification technique handles forms an important subset of transformations that are advocated by the DTSE methodology. These transformations are also widely reported in current practice relating to development of data-intensive software for high-performance and low-power systems. Although our motivation comes from the signal processing application domain, we believe the technique is also useful in other application domains where transformations are applied on source code dominated by indexed variables and iterative control flow.

In future work, we will investigate the extension of our equivalence checking framework to (a) handle algebraic transformations of the arithmetic expressions in the assignment statements; (b) handle data dependencies in control flow and array variable indexing; and (c) further improve the support for error diagnostics. We believe that such an extension will enable verification of nearly all transformations that are relevant for indexed arrays and loops, as applied in the signal processing application domain.

Acknowledgements

The authors thank Eddy De Greef for help with the model extraction tool, Sven Verdoolaege for sharing his array access order checking tool and anonymous referees for constructive comments.

References

- [1] Allen, R., K. Kennedy. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*, Morgan Kaufmann Publishers, 2001.
- [2] Barthou, D., P. Feautrier, X. Redon. On the Equivalence of Two Systems of Affine Recurrence Equations. In the proc. of Euro-Par 2002, LNCS 2400, pp. 309-313.
- [3] Catthoor, F., et. al. *Custom Memory Management Methodology – Exploration of Memory Organization for Embedded Multimedia System Design*, Kluwer Academic Publishers, 1998.
- [4] Čupák, M., F. Catthoor, H. de Man. Efficient System-Level Functional Verification Methodology for Multimedia Applications. IEEE Design and Test of Computers, Vol. 20, No. 2, 2003, pp. 56-64.
- [5] Currie, D.W., A. J. Hu, S. P. Rajan, M. Fujita. Automatic Formal Verification of DSP Software. In the proc. of DAC 2000. pp. 130-135.
- [6] Kelly, W., et. al. The Omega Calculator and Library Version 1.1.0. University of Maryland (1996) Available from: <http://www.cs.umd.edu/projects/omega>
- [7] Nacula, G. C. Translation Validation for an Optimizing Compiler. In the proc. of PLDI, 2000, pp. 83-95.
- [8] Pugh, W. The Omega test: a fast and practical integer programming algorithm for dependence analysis. CACM, 35, 8 (1992), 102-114.
- [9] Samsom, H., F. Franssen, F. Catthoor, H. de Man. System level verification of video and image processing specifications. In the proc. of ISSS, 1995, pp. 144-149.
- [10] Shashidhar, K. C., M. Bruynooghe, F. Catthoor, G. Janssens. An Automatic Verification Technique for Loop and Data Reuse Transformations based on Geometric Modeling of Programs. In the special issue on Compiler Optimization meets Compiler Verification, Journal of Universal Computer Science (J.UCS), Vol. 9, No. 3, 2003, pp. 248-269.
- [11] Zuck, L., A. Pnueli, Y. Fang, B. Goldberg. VOC: A Methodology for the Translation Validation of Optimizing Compilers. J.UCS, Vol. 9, No. 3, 2003, pp. 223-247.